CS 457/557 Functional Programming

Lecture 4
Drawing Shapes
Recall the Shape Datatype

```
data Shape = Rectangle Side Side
            | Ellipse Radius Radius
            | RtTriangle Side Side
            | Polygon [ Vertex ]

  deriving Show
```

```
type Vertex = (Float,Float)
type Side   = Float
type Radius = Float
```
Properties of Shapes

• Note that some shapes are position independent:
  – Rectangle Side Side
  – RtTriangle Side Side
  – Ellipse Radius Radius

• On the other hand, a Polygon [Vertex] is defined in terms of where it appears in the plane.

• A shape’s **Size** and **Radius** are measured in inches.

• On the other hand, the graphics drawing mechanism of Ch. 3 was based on pixels.
Considerations

• Where do we draw position-independent shapes?
  – Randomly?
  – In the upper left corner (the window origin)?
  – In the middle of the window?

• We will choose the last option above, by defining the middle of the window as the origin of a standard Cartesian coordinate system.

• So our new coordinate system has both a different notion of “origin” (middle vs. top-left) and of “units” (inches vs. pixels).

• We will need to define coercions between these two coordinate systems.
Coordinate Systems

Window Coordinate System

Shape Coordinate System

(0,0) pixels
or (1,0) inches

(200,0)

(0,1)

(200,200) pixels
or (1,-1) inches
Units Coercion

\[\text{inchToPixel} :: \text{Float} \rightarrow \text{Int}\]
\[\text{inchToPixel } x = \text{round} \ (100 \times x)\]

\[\text{pixelToInch} :: \text{Int} \rightarrow \text{Float}\]
\[\text{pixelToInch } n = \text{fromIntegral} \ n \ / \ 100\]

Note: simpler alternative to book's definition.
Translation Coercion

\[ x_{\text{Win}}, y_{\text{Win}} :: \text{Int} \]
\[ x_{\text{Win}} = 600 \]
\[ y_{\text{Win}} = 500 \]

\[ x_{\text{Win2}}, y_{\text{Win2}} :: \text{Int} \]
\[ x_{\text{Win2}} = x_{\text{Win}} \div 2 \]
\[ y_{\text{Win2}} = y_{\text{Win}} \div 2 \]

\[ \text{trans} :: \text{Vertex} \to \text{Point} \]
\[ \text{trans} (x,y) = ( x_{\text{Win2}} + \text{inchToPixel} \ x, \]
\[ y_{\text{Win2}} - \text{inchToPixel} \ y ) \]
Translating Points

\[
\text{trans} :: \text{Vertex} \rightarrow \text{Point}
\]
\[
\text{trans} (x, y) = (x\text{Win2} + \text{inchToPixel} \ x, \ y\text{Win2} - \text{inchToPixel} \ y)
\]

\[
\text{transList} :: [\text{Vertex}] \rightarrow [\text{Point}]
\]
\[
\text{transList} [] = []
\]
\[
\text{transList} (p:ps) = \text{trans} \ p : \text{transList} \ ps
\]

-- or:
\[
\text{transList} \ vs = [\text{trans} \ p \mid p <- \ vs]
\]
Translating Shapes

\[
\text{shapeToGraphic} :: \text{Shape} \rightarrow \text{Graphic} \\
\text{shapeToGraphic} \ (\text{Rectangle} \ s1 \ s2) \\
\quad = \ \text{let} \ s12 = s1/2 \\
\quad \quad \quad \quad s22 = s2/2 \\
\quad \quad \text{in polygon} \\
\quad \quad \quad \quad (\text{transList} \ [(-s12,-s22),(-s12,s22),\ (s12,s22),\ (s12,-s22)]) \\
\text{shapeToGraphic} \ (\text{Ellipse} \ r1 \ r2) \\
\quad = \ \text{ellipse} \ (\text{trans} \ (-r1,-r2)) \ (\text{trans} \ (r1,r2)) \\
\text{shapeToGraphic} \ (\text{RtTriangle} \ s1 \ s2) \\
\quad = \ \text{polygon} \ (\text{transList} \ [(0,0),(s1,0),(0,s2)]) \\
\text{shapeToGraphic} \ (\text{Polygon} \ pts) \\
\quad = \ \text{polygon} \ (\text{transList} \ pts)
\]

Note: first three are position independent and centered about the origin
Some Test Shapes

sh1, sh2, sh3, sh4 :: Shape

sh1 = Rectangle 3 2
sh2 = Ellipse 1 1.5
sh3 = RtTriangle 3 2
sh4 = Polygon [(-2.5,2.5), (-1.5,2.0),
                 (-1.1,0.2), (-1.7,-1.0),
                 (-3.0,0)]
main10
    = runGraphics (  
        do w <- openWindow "Drawing Shapes" (xWin,yWin)
          drawInWindow w
            (withColor Red (shapeToGraphic sh1))
          drawInWindow w
            (withColor Blue (shapeToGraphic sh2))
          spaceClose w
    )
The Result
Drawing Multiple Shapes

type ColoredShapes = [(Color,Shape)]

shs :: ColoredShapes
shs = [(Red,sh1),(Blue,sh2),
       (Yellow,sh3),(Magenta,sh4)]

drawShapes :: Window -> ColoredShapes -> IO ()
drawShapes w [] = return ()
drawShapes w ((c,s):cs)
  = do drawInWindow w
       (withColor c (shapeToGraphic s))
       drawShapes w cs
Multiple Shapes, cont’d

main11
  = runGraphics ( 
    do w <- openWindow
       "Drawing Shapes"
       (xWin,yWin)
       drawShapes w shs
       spaceClose w
     )
Retrospect

• Can distinguish three different types.

```haskell
data Shape = Polygon [Vertex] | ... 
```

» “Transparent” -- can both construct and pattern match.
» Represents geometric abstraction (no graphical meaning)

```haskell
type Graphic
polygon :: [Point] -> Graphic 
```

» Abstract type – can construct instances, but not inspect them.
» Can modify/combine with special operators like `withColor`
» Expressed in graphics coordinate system.

```haskell
type IO ()
drawPolygon :: [Vertex] -> IO () 
```

» (We didn't choose to define functions like this.)
» Even more abstract; can only be sequenced and executed.